# AFRL-IF-WP-TR-2002-1512


# ADA/PORTABLE OPERATING SYSTEM INTERFACE (POSIX) MATURATION
## DELIVERY ORDER 0004: AVIONICS SOFTWARE TECHNOLOGY SUPPORT (ASTS)

**Marc J. Pitarys**

**EMBEDDED INFORMATION SYSTEM ENGINEERING (AFRL/IFTA)**
**INFORMATION TECHNOLOGY DIVISION**
**INFORMATION DIRECTORATE**
**AIR FORCE RESEARCH LABORATORY, AIR FORCE MATERIEL COMMAND**
**WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334**

**HUGHES AIRCRAFT COMPANY (RAYTHEON)**
**2000 E. IMPERIAL HIGHWAY**
**EL SEGUNDO, CALIFORNIA 90245**

**NOVEMBER 1995**

**FINAL REPORT FOR PERIOD 23 JANUARY 1995 – 30 NOVEMBER 1995**



**INFORMATION DIRECTORATE**
**AIR FORCE RESEARCH LABORATORY**
**AIR FORCE MATERIEL COMMAND**
**WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334**

# NOTICE

Using government drawings, specifications, or other data included in this document for any purpose other than government procurement does not in any way obligate the U.S. Government. The fact that the government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey and rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report has been reviewed by the Office of Public Affairs (ASC/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

MARC J. PITARYS, Project Engineer
Embedded Information System Engineering Branch
AFRL/IFTA

JAMES S. WILLIAMSON, Chief
Embedded Information System Engineering Branch
AFRL/IFTA

FOR
EUGENE C. BLACKBURN

EUGENE C. BLACKBURN, Chief
Information Technology Division
AFRL/IFT

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. REPORT DATE *(DD-MM-YY)* | 2. REPORT TYPE | 3. DATES COVERED *(From - To)* |
|---|---|---|
| November 1995 | Final | 01/23/1995 – 11/30/1995 |

**4. TITLE AND SUBTITLE**
ADA/PORTABLE OPERATING SYSTEM INTERFACE (POSIX) MATURATION DELIVERY ORDER 0004: AVIONICS SOFTWARE TECHNOLOGY SUPPORT (ASTS)

**5a. CONTRACT NUMBER**
F33615-92-D-1050

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**
78611F

**6. AUTHOR(S)**
Marc J. Pitarys (AFRL/IFTA)

**5d. PROJECT NUMBER**
3090

**5e. TASK NUMBER**
01

**5f. WORK UNIT NUMBER**
18

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

Embedded Information System Engineering (AFRL/IFTA)
Information Technology Division
Information Directorate
Air Force Research Laboratory, Air Force Materiel Command
Wright-Patterson Air Force Base, OH 45433-7334

Hughes Aircraft Company (Raytheon)
2000 E. Imperial Highway
El Segundo, CA 90245

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

**10. SPONSORING/MONITORING AGENCY ACRONYM(S)**
AFRL/IFTA

**11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)**
AFRL-IF-WP-TR-2002-1512

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT** *(Maximum 200 Words)*

This effort matures and demonstrates Portable Operating System Interface (POSIX) features for strike fighter avionics software. The maturation and demonstration focused on assessing the performance and security of selected POSIX interfaces. The Hughes Aircraft corporation Avionics Operating System (AOS) was used as benchmark to compare the performance and security results from the POSIX assessment.

**15. SUBJECT TERMS**
Ada, POSIX, System application programming interface (API)

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT: | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON (Monitor) |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | Marc J. Pitarys |
| Unclassified | Unclassified | Unclassified | SAR | 34 | 19b. TELEPHONE NUMBER *(Include Area Code)* (937) 255-6548 x3608 |

**Standard Form 298 (Rev. 8-98)**
Prescribed by ANSI Std. Z39-18

## Table of Contents

## Abstract

The goal of this effort is to mature and demonstrate Portable Operating System Interface (POSIX) features for strike fighter avionics software. The maturation and demonstration focused on assessing the performance and security of selected POSIX interfaces. The Hughes Aircraft Corporation Avionics Operating System (AOS) was used as a benchmark to compare the performance and security results from the POSIX assessment.

The Ada/POSIX Maturation (APM) effort implemented, demonstrated, and evaluated the latest drafts of Ada bindings to the POSIX real-time standards. The AOS was extended to support the new POSIX application programming interfaces (APIs). The real-time features of POSIX and their performance characteristics were measured and analyzed by executing a test suite that contained these features.

The analysis of POSIX for secure time critical avionics systems helped quantify and reduce the risks of adopting POSIX for new strike weapon systems. The acceptance of a standard operating system API is critical when re-using software artifacts since the operating system and its API define the foundation for the application software architecture. Specifying the API early in a program facilitates the re-use of software developed during the concept demonstrations of the operational system. Also, significant investments in existing avionics applications can be leveraged when the software architecture and the specified API are backward-compatible.

# 1.0 Introduction

## Statement of the Problem

Software development and post deployment support costs are a significant portion of an avionics system's life cycle costs. Increases in software productivity through the re-use of existing avionics domain software necessitate the acceptance of standard software interfaces. The standardization of the operating system interface facilitates the re-use of application code, tests, designs, architectures, specifications, and knowledge as the operating system interface is reflected in each of these. Software development and maintenance are also improved by the commonality across avionics subsystems, platforms and military services.

Advanced military aircraft objectives demand security and real-time characteristics from avionics operating systems. These characteristics are not assured in POSIX as the standard is still quite new and implementations of critical components such as real-time extensions have only just begun to appear. The standardization of Ada bindings to POSIX has lagged the C language bindings. POSIX was originally intended as a set of standard interfaces to UNIX-like operating systems and has never been used for an avionics system. Therefore, the specification of POSIX for an avionics program involves cost, schedule and performance risks. The reduction of these risks are made possible through the evaluation and demonstration of the security and real-time characteristics of the API and its implementation.

**NOTE**: Throughout this report, references to various Institute of Electrical and Electronic Engineers (IEEE) POSIX standards are made with the shorthand "POSIX.*n*", which means IEEE POSIX Standard 1003.*n*. See section 4.0, "References", for the complete titles of the standards.

## Problem Attack

The task of evaluating the suitability of POSIX to advanced avionics systems requires an analysis of the relative performance, security features, and difficulty of implementing the POSIX Ada bindings, i.e., complexity. This can only be accomplished through the implementation, demonstration, and evaluation of a proven advanced avionics operating system that incorporates the relevant POSIX functionality.

For the APM effort, Hughes leveraged the AOS. The AOS, developed by Hughes, is used in over one million lines of avionics application code. The AOS API is currently being used on F-18 and F-22.

Hughes identified the relevant profile of POSIX Ada bindings, implemented those bindings in the AOS, and demonstrated a comparison of AOS API and POSIX API performance. The results of this approach assisted in determining the feasibility of using an Ada POSIX operating system in an embedded avionics system. In addition, a better understanding of the benefits and risks associated with POSIX based on detailed facts and data were accomplished.

## Tasks Completed For This Study

Four tasks were undertaken for this study:

### Task 1: Select Candidate POSIX Features.

The POSIX standards are intended to support a very broad selection of platforms including supercomputers, mainframes, workstations, personal computers, and embedded systems. The POSIX working groups have identified and produced standards for various environments, some of which are not necessary or useful in avionics systems. POSIX.5b and the dedicated real-time profile (POSIX.13, PSE52) were reviewed and evaluated to identify the subset of the POSIX Ada bindings to be implemented in the AOS.

### Task 2: POSIX Avionics Operating System Development.

The AOS was modified to support the POSIX subset identified in Task 1. The AOS maintained backward compatibility from the AOS API where feasible and where it did not interfere with the demonstration in Task 3. A test suite was then created to exercise and time the comparable features of the two APIs in functionally equivalent ways and report the results.

### Task 3: POSIX Avionics Operating System Demonstration.

Hughes demonstrated the functionality and performance of the POSIX subset. The performance characteristics of the POSIX API were compared with an AOS API baseline. The demonstration depicted system service execution times, performance and sizing information for each feature of the POSIX subset. In addition, a security impact, and a description of any implementation difficulties encountered during Task 2 were documented.

### Task 4: POSIX Assessment and Report.

The report evaluates the feasibility of using an Ada POSIX operating system. The functional capabilities and performance of the POSIX subset were summarized. Any performance shortfalls, inadequate or immature POSIX standards, security issues, and implementation difficulties were cited.

## Approach

The POSIX interfaces features to be demonstrated were selected based on the following criteria:

- The POSIX services are from those described in the most recent draft of POSIX.5b.
- The AOS must support the same or functionally equivalent services to provide a basis for comparison. For example, this eliminates features such as Shared Memory (which the AOS lacks) or Global Bulk Memory Services (which POSIX lacks.)
- The services should be commonly used by applications. Demonstration of esoteric features would prove little about the feasibility of using POSIX in an avionics system.

- The functionality should be present in the POSIX.13 Dedicated Real-time Profile. This is the "avionics" profile.
- For the most part, the services should be time-critical.

Based on these criteria and funding constraints, the following features of POSIX were selected to be implemented:

- Semaphores
- Timers
- Event Notification (POSIX Real-Time Signals and AOS Event Clusters)
- Interprocess Communication (POSIX Message Queues and AOS Labeled Messages)

## Resources & Tools Used

The AOS microkernel, called the Functional Core, was modified to execute on the DMV (DeMonstration Vehicle), a Hughes i960-based breadboard. This platform was used in the initial development of the AOS before the Common Integrated Processor (CIP) hardware was available. This board runs at 16 MHz, and supports 2 Mbytes of static RAM.

The Irvine Compiler Corporation (ICC) i960 MX Ada Compilation System (version 7.7.5B) was used to compile and link the AOS POSIX implementation and test suite. Both the AOS and the POSIX interfaces were implemented in Ada, supplemented by insertions from package `Machine_Code` where necessary.

## 2.0 Data

For each POSIX service, data was collected for Timings, Executable Size, Complexity, Security, and Implementation Difficulties.

Timings were collected in an effort to discover if there were any insurmountable problems to achieving acceptable performance in the semantics of the POSIX services. The timing results of the features identified in section 1.0 are detailed in the *Service Timings* section for each service detailed in section 2.0. The majority of metrics collected were based on the suggestions in POSIX.1b, Annex G—"Performance Metrics".

Executable Size was measured to determine if implementing the POSIX subset would result in unacceptable memory consumption, since memory is usually a scarce resource on embedded targets.

Complexity was measured by a Source Lines of Code (SLOC) count. This measurement relates to cost, as more SLOC means more time to produce the code.

An analysis of the security impacts of the POSIX services was performed to discover where security problems may exist and how they may be remedied. Also, the impact on the POSIX services by the interfaces that implement security was examined.

Lastly, any implementation difficulties seen during the implementation of the POSIX interfaces were summarized.

## Service #1: Semaphores

Semaphores are a low-level synchronization primitive that are used by tasks or processes to synchronize their execution or to insure mutual exclusion when accessing shared data structures. A semaphore is an integer variable associated to a group of suspended tasks. The state or value of a semaphore can be accessed and altered only at creation/initialization and by the `Lock` and `Unlock` operations. *Binary* semaphores can only assume the value 0 or 1. *Counting* semaphores can assume non-negative integer values (although some implementations of semaphores let the count go negative to indicate the number of suspended tasks).

The `Lock` operation on a semaphore *S* is as follows:

```
if S > 0 then
    S := S - 1;
else
    suspend on S;
end if;
```

The `Unlock` operation on a semaphore *S* is :

```
if (one or more tasks are waiting on S) then
    let one of these tasks proceed;
else
    S := S + 1;
end if;
```

Both `Lock` and `Unlock` are atomic operations. Both POSIX and AOS support counting semaphores.

POSIX defines two types of counting semaphores: *anonymous* and *named.* The operations (`Lock` and `Unlock`) on theses two types of semaphores are the same once the semaphore exists; the difference is in how the semaphore is created and destroyed. Named semaphores have a name (text string) associated with them and are opened, closed and unlinked in a manner similar to a file. Named semaphores allow the sharing of a semaphore between unrelated processes (that is programs). Anonymous semaphores have no name (hence the term) and are created and destroyed by calls to `Create` and `Destroy` rather that `Open` and `Close`. Anonymous semaphores may also be shared between any processes that have access to the semaphore if the Process Shared attribute of the semaphore is set.

The AOS only supports anonymous semaphores. Since the AOS was created for a system where all the programs running are written in Ada, it was believed that multitasking would be achieved though the use of Ada tasks, not multiple cooperating programs. Since tasks within a single Ada program share the same address space, they can utilize anonymous semaphores. Named semaphores were therefore considered unnecessary. In addition, the AOS does not allow the sharing of anonymous semaphores between processes.

## Service Timings

As shown in Figure 2.1.1, the difference in service timings between the AOS and POSIX API implementations is negligible. This is to be expected; the specification of the operations on POSIX semaphores doesn't differ greatly from AOS semaphore operations.

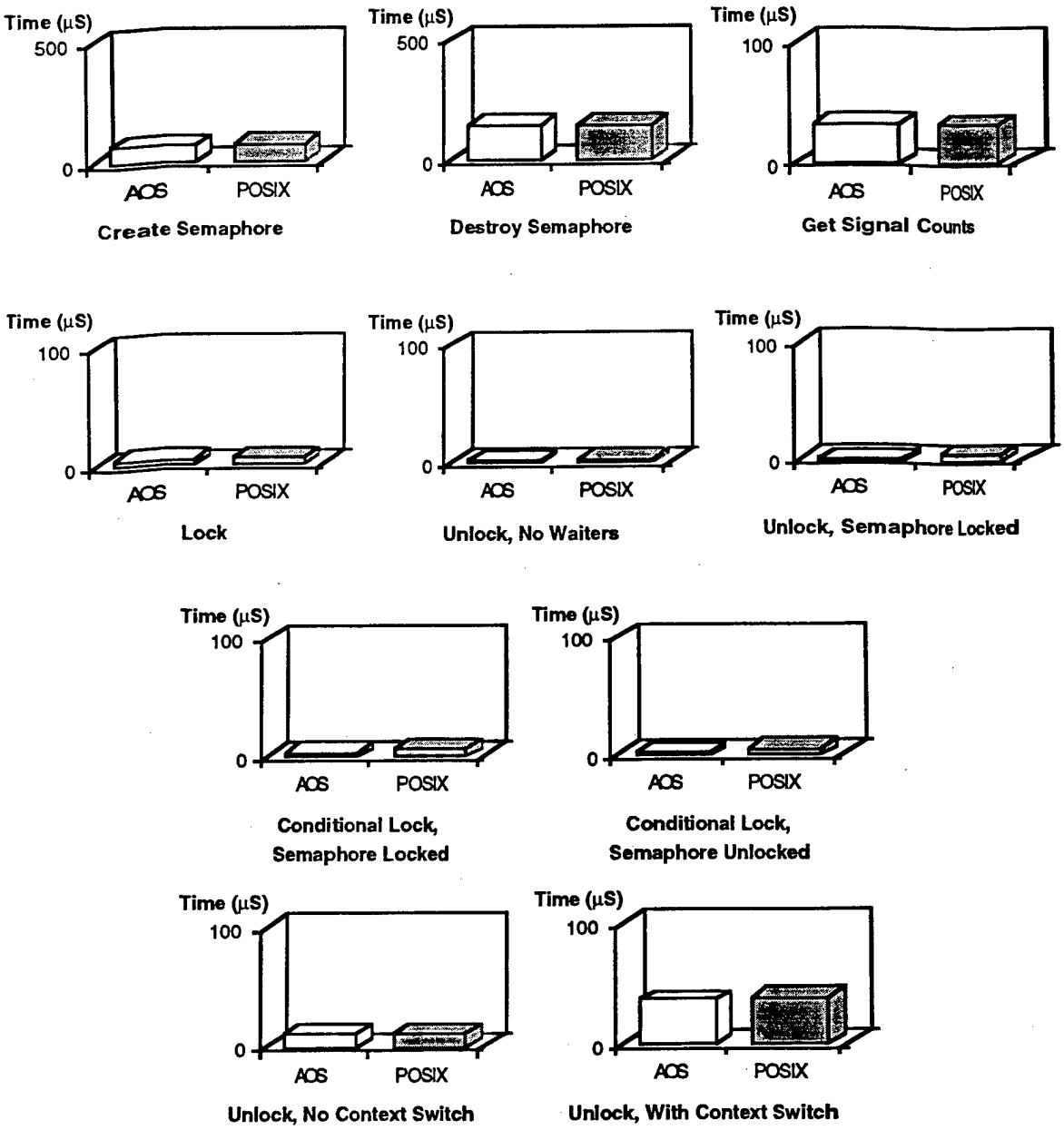



**Figure 2.1.1:** Semaphore Timings

## Executable Size

As shown in Figure 2.1.2, the executable size of the POSIX semaphores was 2.4 times as large as AOS semaphores due to the additional functionality of named

semaphores. Essentially, all of the size difference between the two implementations is due to the named semaphores functionality. Overall, the size of POSIX anonymous semaphores is the same as the AOS implementation.

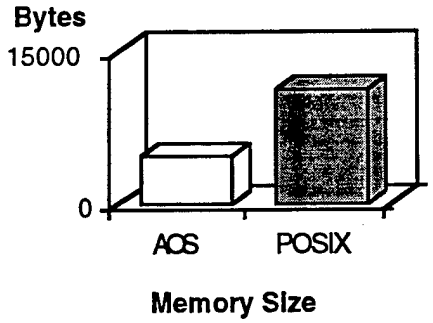



**Figure 2.1.2:** Semaphore Executable Size

### *Complexity*

Since POSIX semaphores defines additional functionality, it follows that its SLOC count would be greater than the AOS. The POSIX implementation was approximately 1.9 times as large as the AOS, as shown in Fig. 2.1.3.

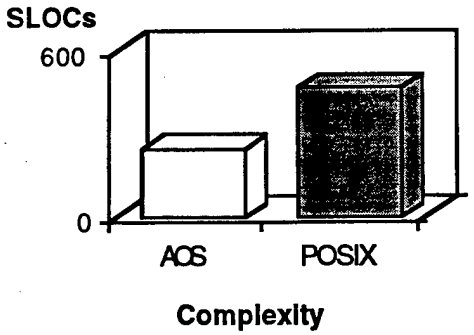



**Figure 2.1.3:** Semaphore Statement Lines of Code (SLOC)

### *Security*

POSIX allows the sharing of semaphores between processes. This allows the semaphore count to be used as a covert channel. A covert channel is an unsanctioned communication path that could be used to transfer data illicitly. To prevent this, access to the semaphores needs to be mediated and restricted by the OS.

Restricting access to semaphores has several facets. The first concerns named semaphores. Access must be checked on an `Open` of a named semaphore to insure that the calling process has the necessary privilege to access the semaphore. The POSIX implementation must also ensure that semaphore descriptors (the abstract handle that the OS gives back to the application on an `Open`) cannot be propagated (via interprogram communication or shared memory) to other processes that do not have the privilege to access the semaphore. This requires that the semaphore descriptor only be valid within the process that originally received the semaphore descriptor from the OS. This same

restriction must be applied to anonymous semaphores as well. Therefore, for security concerns, the Process Shared attribute of POSIX anonymous semaphores must always be set to false.

The restriction that a semaphore descriptor is only valid within the process that originally received the semaphore descriptor from the OS does not apply to forked child processes. A forked child process is an exact duplicate of the parent that spawned it; therefore it is reasonable that the child should be able to access any data that the parent has access to.

### *Implementation Difficulties*

There were no unusual difficulties in implementing POSIX Semaphores.

## Service #2: Timer Services

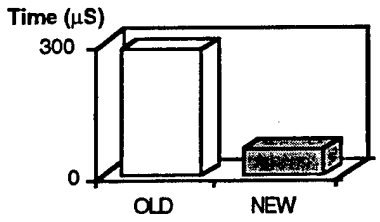
The AOS and POSIX have similar services to read/set the system clocks and to create and arm both absolute and relative interval timers based on those clocks.

### *Service Timings*

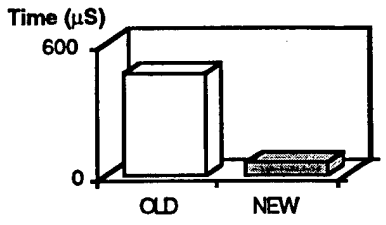
The service times for the creation of a timer, the reading of the system clock, and the disarming of timers (both armed and quiescent) were quite similar on the two implementations, differing by only a few microseconds. However, in the initial implementation of POSIX timers, the amount of time needed to arm a timer was far greater than AOS, taking as much as 5 to 10 times as long. This was determined to be caused by how the POSIX time type (`POSIX.Timespec`) was specified.

The draft 4.0 version of POSIX.5b specified that `Timespec` should have both a precision and a resolution of nanoseconds. On the DMV target, the system clock is a 64 bit microsecond timer. Hence, on an arm timer request each parameter of type `Timespec` needed to be converted to the underlying timer hardware representation. This conversion from nanoseconds to microseconds required 64 bit division on a target machine that did not directly support this. This conversion caused the additional timing overhead.

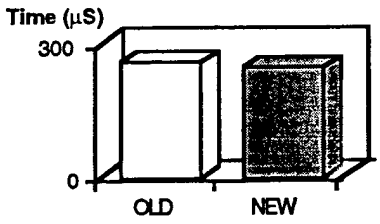
After reporting the problem with the POSIX `Timespec` at the January 1995 IEEE POSIX Working Group meeting, the technical editors of POSIX.5b decided that it was not necessary to require the resolution of the underlying implementation of `Timespec` to be any greater than the required resolution of the system clock. This change will allow system implementors to choose the implementation most appropriate for their hardware configuration. Figure 2.2.1 illustrates the change in performance between the two implementations. Overall, this problem with the POSIX Timespec implementation demonstrated the need to conduct a "hands-on" analysis of standard operating system features before they are incorporated into an avionics operating system.



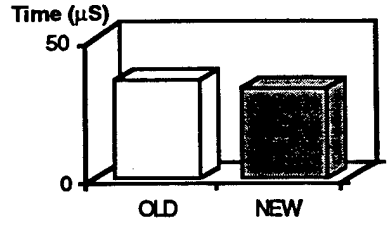

**Arm Absolute Timer**




**Arm Absolute Timer, Periodic**




**Absolute Timer Expiration Latency**




**Disarm Absolute Timer**

There was also significant difference in the latency of notification of timer expirations between the AOS and POSIX timers. This is the difference between the time that a timer was set to expire and the time that the first instruction is executed in the task waiting for the timer expiration. The POSIX latency was greater by slightly more than a factor of two, primarily due to the greater overhead of signal delivery (the POSIX method of asynchronous notification) as opposed to the equivalent AOS method of signaling an event cluster. Figure 2.2.2 compares the AOS and improved POSIX timings. (Recall that these POSIX timings were improved as a result of the POSIX Working Group's decision not to require the resolution of the underlying implementation of Timespec to be any greater than the required resolution of the system clock.)

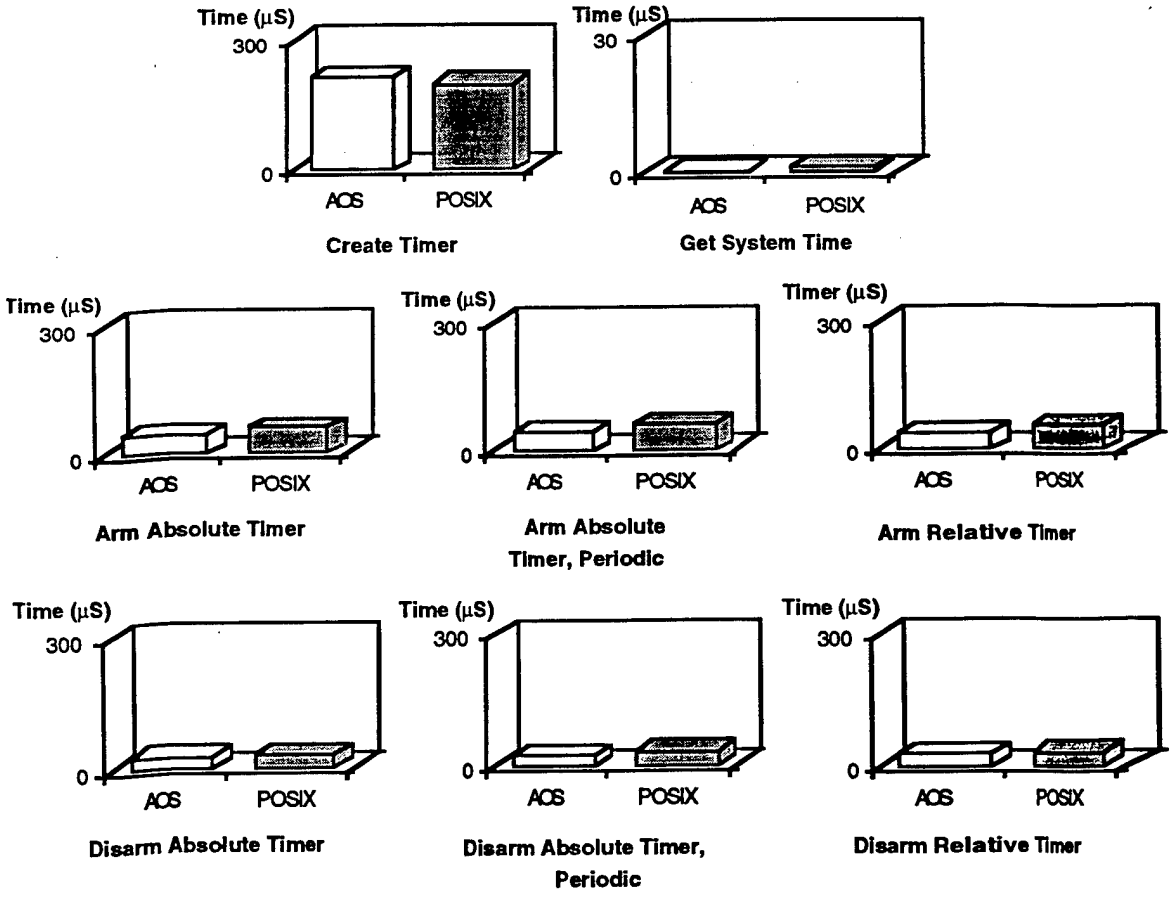



**Figure 2.2.2:** Timer Timings

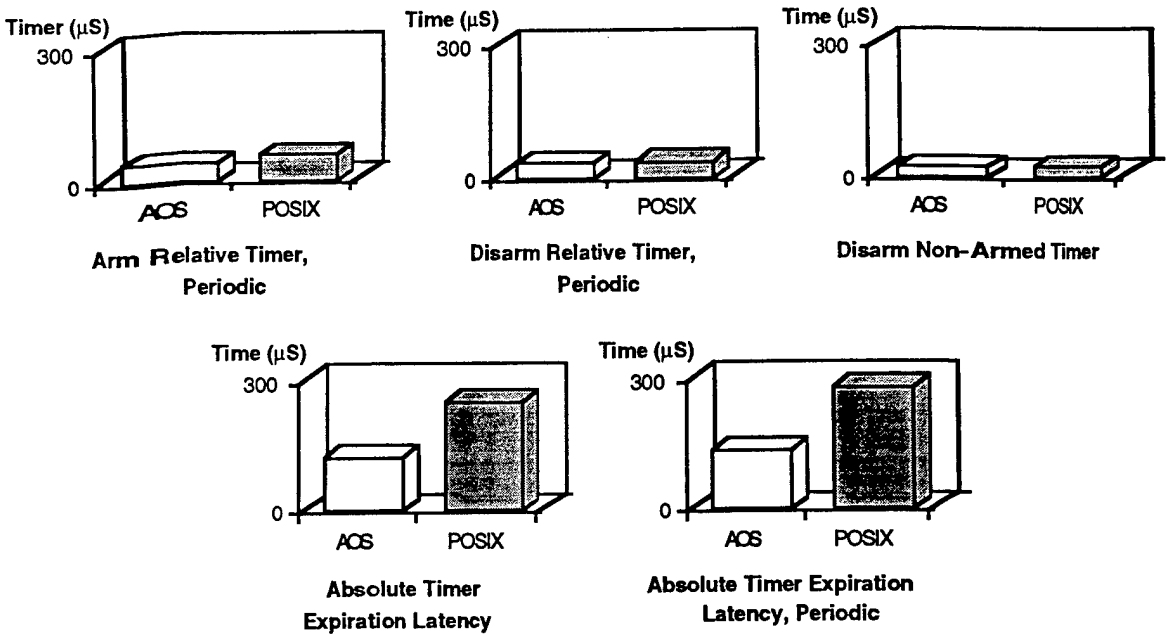



**Figure 2.2.2 (cont'd):** Timer Timings

## *Executable Size*

The AOS timers were 1.5 times as large as the POSIX timers, as shown in Figure 2.2.3. This is due to additional functionality provided by the AOS interface. The AOS allows the caller to specify one or more notification mechanisms (semaphores, event clusters, or status blocks); this requires additional code to implement. The AOS also provides services to suspend and resume all armed timers.

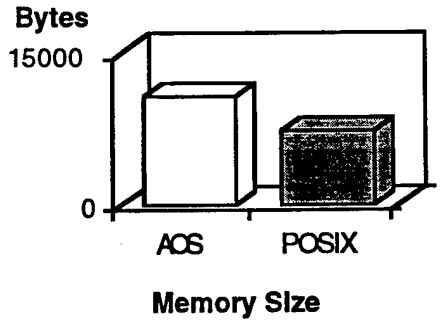



**Figure 2.2.3:** Timer Executable Size

## *Complexity*

As shown in Figure 2.2.4, the AOS timer implementation SLOC count is 1.3 times as large as the POSIX implementation owning to the additional functionality mentioned in above in *Executable Size*.

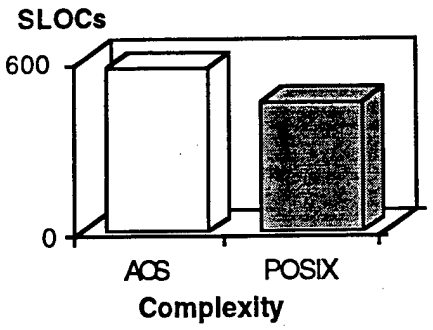



**Figure 2.2.4:** Timer Statement Lines of Code (SLOC)

### *Security*

Since timer requests are contained within a single process, there are no security issues regarding their use. The one time related service that may have security implications is the setting of the system clock (or any clock that another process may read). The POSIX standard does allow an implementation to restrict the setting of the system clock to those with some implementation-defined "appropriate privilege."

### *Implementation Difficulties*

The one major thorn in the side of implementors will be the arithmetic operations on `POSIX.Timespec`. In POSIX.1b, the structure `Timespec` was defined as a C structure with two members: `tv_sec` (a signed seconds count) and `tc_nsec` (an unsigned nanoseconds count). This data structure appears in the Ada binding as the private type `POSIX.Timespec`. (It was decided to make the type private to allow greater freedom to implementors to utilize the best representation for the type.) Package POSIX defines the arithmetic operations on `Timespec`, conversions operations to and from `Standard.Duration`, as well as services to get and set the seconds and nanoseconds attributes of `Timespec` variables.

However, an oddity about how `Timespec` represents negative time values makes some of these operations difficult to implement. If the seconds attribute is signed, but the nanoseconds attribute is unsigned, then the representation for -0.6 seconds must be (seconds = -1, nanoseconds = $0.4 \times 10^9$). This leads to the following problem: If the data representation of `Timespec` follows the C interface, the conversions to and from seconds, nanoseconds and `Timespec` become trivial to implement, but the arithmetic operations become quite complicated and (perhaps more importantly) slow. If the implementation of `Timespec` is a simple signed long integer count, the arithmetic operations become less complicated, but the complexity of the conversion operations increases substantially. In either case, the implementation has a performance penalty.

## Service #3: Event Notification

POSIX currently has the delivery of signals as its only form of asynchronous notification (that is notification of asynchronous events such as timer expiration, asynchronous I/O completion, etc.). The POSIX real-time extensions adds the concept of *real-time* signals, which provide for queuing (normal signals are not normally queued,) and the attaching of data to a signal occurrence. Real-time signals are used as follows: each type of service that needs to provide an asynchronous notification of completion takes as a parameter a value of `POSIX_Signals.Signal_Event`. This data structure contains at least the following pieces of information:

- which signal to send when the event of interest occurs
- the notification type [currently either `Signal_Notification` (send the specified signal) or `No_Notification` (do nothing)]
- an optional piece of user data to be queued along with the signal.

The user can then call one of several services that allow the calling task to be suspended until one of a set of specified signals occur, or (optionally) until a time-out occurs.

AOS semaphores and event clusters serve the same function that POSIX real-time signals do. The services that require asynchronous notification allow the user to provide a semaphore or event cluster (or both) and an optional status block. When the event of interest occurs, the provided notification mechanisms will be updated. The calling task may poll on the status block, or suspend itself on the semaphore or event cluster waiting for the event of interest to occur.

### *Service Timings*

The POSIX services to send or await a signal occurrence were compared to the AOS event clusters, since event clusters are more functionally equivalent to POSIX real-time signals than semaphores are. As noted in the timer expiration latency times, the time to send or receive a POSIX signal is significantly greater than the equivalent AOS time to signal an event cluster, or receive notification that an event cluster has been signaled (by approximately a factor of 2.) Low latency in responding to events is an important aspect of real-time systems, making this long latency a significant problem.

The somewhat complex semantics of signal delivery in POSIX is the main cause of the longer latency. The processing to send a signal involves checking which signal is being sent (some signals such as `Signal_Null` and `Signal_Kill` have special semantics), whether the signal is currently ignored or blocked from delivery, etc. Overall, signals are a high-overhead means of asynchronous notification of events. Allowing for the use of lower over-head synchronization mechanisms (such as semaphores or condition variables) in asynchronous notification would seem to be a reasonable addition to the real-time POSIX standards.

It should be noted that the original specification of real-time signals was targeted at single-threaded C programs. In such an environment, a relatively efficient implementation of real-time signals can be realized. This is due to the fact that there can never be more than one thread of control waiting for the signal to occur. When the notification signal is sent, there is no question of which thread is waiting for the signal since there is only one thread.

However, in a multi-threaded environment (whether in C or Ada) the implementation becomes far more complex. It is possible to have multiple tasks suspended, waiting for a possibly overlapping set of signals. This added difficulty is the other component of the longer latency times recorded for POSIX signals.

There are two basic multi-threaded implementations for real-time signals. The first implementation has all the tasks in one long queue. When a signal is delivered, the OS must search the queue for the first task which is waiting for that signal. The time to deliver a signal is a function of how many tasks are currently suspended waiting for signal notifications, and the order in which they were suspended. This cannot generally be predicted by the application, making timing analysis difficult. This approach is quite simple to implement, but has the disadvantage that it is non-deterministic, and thus unsuitable for real-time systems.

The second implementation has a queue of tasks for each signal that can be waited for, and when a task suspends waiting for a set of signals, it must be placed on multiple queues, one for each signal in the set. (For example, if the task were waiting for signals *X*, *Y*, and *Z*, the task would be placed on the queues associated with signals *X*, *Y*, and *Z*.) When a signal is delivered, the OS simply dequeues the task at the head of the queue associated with the delivered signal; there is no search involved. If the task is waiting for *n* signals, the OS must also remove the task from the *n*-1 other queues that it is on. However, if some care is exercised in setting up the data structures, this can be performed in a manner such that the amount of time required to do this is a linear function of the number of queues that must altered. This makes the time to deliver a signal a function of how many signals the task is simultaneously waiting for (the time should increase a fixed amount for each additional signal in the set.) If the task is waiting for only one signal, the time to deliver the signal should be essentially constant. This will allow an application to account for the event notification overhead in their timelines.

Figure 2.3.1 illustrates the timing data for POSIX Signals.

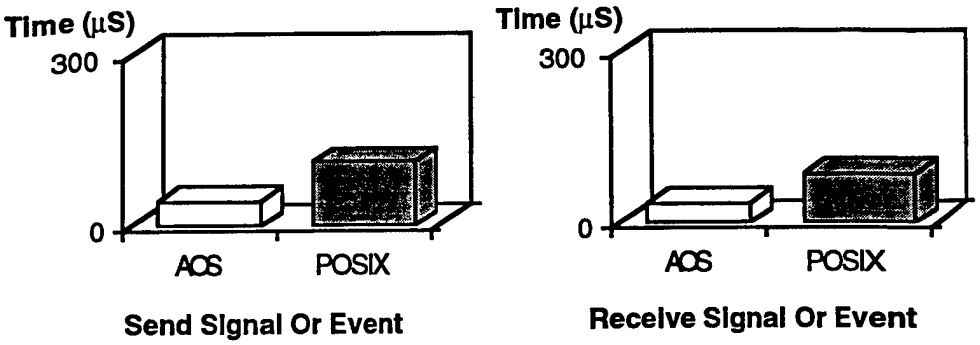



**Figure 2.3.1: Signal Timings**

### *Executable Size*

As shown in figure 2.3.2, the size of the POSIX implementation was 1.6 times as great as the AOS event cluster mechanism. This is due to the more complex semantics associated with POSIX signals. Note that this size comparison compares only static code and data structures; it does not account for the dynamic memory required for the notifications and their linking onto either POSIX or AOS data structures.

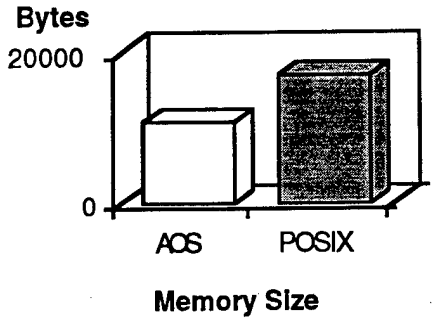



**Figure 2.3.2:** Signal Executable Size

### *Complexity*

As shown in Figure 2.3.3, the SLOC count for the POSIX signals was 2.2 times as great as the AOS event clusters, and again this is due to the more complex semantics associated with POSIX signals.

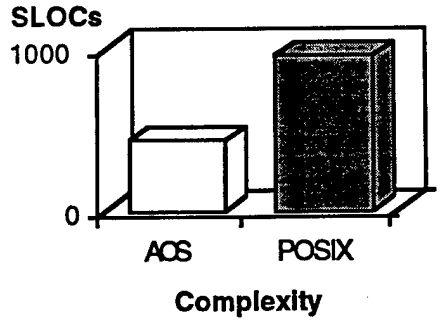



**Figure 2.3.3:** Signal Statement Lines of Code (SLOC)

### *Security*

POSIX signals as they are used for asynchronous event notification do not present a security problem since the process receiving the signal is the same one that requested the notification. A data flow from a process to itself is always permissible (and in fact, impossible to prevent). However, POSIX signals can be used for interprocess communication (AOS event clusters can only be used for intraprocess communication) and thus are a data channel that must be mediated by the OS. POSIX.1e addressed these issues, and proposes several changes and additions to the semantics of sending a signal.

The first addition assumes that each process has an associated Mandatory Access Control (MAC) label. When sending a signal between processes, the label of the sending and receiving process must be compared according to the following rules:

(1) If the MAC label of the sending process is equivalent to the MAC label of the receiving process, then no additional requirements are posed.

(2) If the MAC label of the sending process dominates the MAC label of the receiving process (i.e., the signal is being "written down"), then the sending process must have appropriate privilege.

(3) If the MAC label of the sending process is dominated by the MAC label of the receiving process, (i.e., the signal is being "written up"), then it is implementation-defined whether the sending process requires appropriate privilege.

(4) If the MAC label of the sender and the receiver are incomparable, then the sending process must have appropriate privilege.

This checking would take place during what might be considered a time-critical service and would therefore have an impact on performance. The AOS design has attempted to confine such checking to non-time-critical "connect" or setup services. This one-time check determines if the caller has the appropriate privilege for the requested capability. (If the caller does, a capability token is passed back to the caller.) This token is then passed back to the OS during time-critical services as proof that the caller has the appropriate privilege to perform certain tasks. This method however, will not work in the case of signals since it is not known until a signal is sent who the receiving process(es) will be.

A second change involves the use of `Signal_Null`. If `Signal_Null` is sent to a process, no actual signal is sent, but all the necessary error checking takes place. This can be used to determine if a process with a certain process ID exists. POSIX.1e specifies that if the MAC label of the sending process does not dominate the MAC label of any receiving process and the sender does not have appropriate privilege, then the system returns an error indicating that the receiving process does not exist.

The last change involves granting the capability for processes to block certain "unblockable" signals (such as `Signal_Kill`).

### *Implementation Difficulties*

The complex semantics associated with POSIX signals make their implementation challenging; however, there were no unexpected difficulties. Yet, there is one element of the POSIX real-time signal delivery model that bears discussion.

In order to queue a signal for delivery, a POSIX process must utilize some system resources (mainly, memory for the data). POSIX places a resource limit on how many queued signals a process may send and simultaneously have pending at receiving processes. A difficulty arises from the fact that this limit is not how many signals the process may itself have pending (i.e., signals it is receiving) but how many it can send. This complicates reclaiming these resources when the sending process terminates and still has signals pending at other processes. Since the resources being used to queue the sent signals are still in use by the receiving process(es), they cannot be reclaimed during the sending process' termination. (There is no rationale in the POSIX standards for making this limit the number of queued signals that can be simultaneously sent versus received. One explanation is that if the limit were a receive limit, one process could monopolize another process' signal queuing resources by repeatedly sending a signal to that process, effectively preventing reception of queued signals from other sources.)

## Service #4: Message Queues

POSIX Message Queues provide services for interprogram communication. A message queue is a collection of messages, each message being an array of byte values with an associated priority. Within the message queue, the messages are kept in priority order with first-in, first-out order among messages with the same priority. It is left to the application to impose a data structure upon a message.

POSIX Message Queues are accessed by applications in a way similar to files. A queue is opened (and optionally created if it does not exist) using a POSIX pathname. `Send` and `Receive` operations enqueue and dequeue messages from a queue. Once created, a queue's name has global scope, and the queue is accessible by all processes having the proper access permissions. Any number of processes may have a queue open, within system configuration limits, for reading, writing, or both.

The AOS Labeled Message services also provide for interprogram communications, but use a very different approach and mechanism. This makes a direct comparison very difficult. In the AOS, Labeled Messages are known and statically declared at compile time, and contain the destination embedded in the message. Buffer management is left to the application. Different queues deal with different priorities of messages which are statically known, and thus can be statically created. With POSIX Message Queues, messages can be dynamically created and contain only data. Processes that wish to communication must `Open` the same queue via the same pathname, which again, may be dynamically determined, so queues must be dynamically created.

Finally, the Labeled Message services also support distributed inter-processor communications while Message Queues do not. Thus, the Executable Size and Complexity comparisons have been adjusted to discount interprocessor communications.

### *Service Timings*

The AOS Labeled Message services and mechanism were designed to perform as much verification and validity checking as possible when the application is built instead of at run-time. The run-time services themselves need perform little or no checking. The POSIX Message Queue services on the other hand have no such build-time checking, and all validity checks must be performed at run-time. This results in the nearly double the overhead in a POSIX `Send` and `Receive` operations versus the AOS Transfer (the actual AOS subprogram is called `Send`, but it performs both the send and receive operations in one step.)

The AOS Labeled Message mechanism also allows the queues to be statically created, since the number of queues is based only on the different message priorities available. Thus, no "queue creation" is necessary and the time to open a channel for a message is always repeatable. The POSIX Message Queue services determines the destination of a message (the queue) when the queue is opened or created, and thus may need create new queues at run-time. This is apparent in the rather large `Open_Or_Create` time for a new queue, but a very short `Open_Or_Create` time for an existing queue.

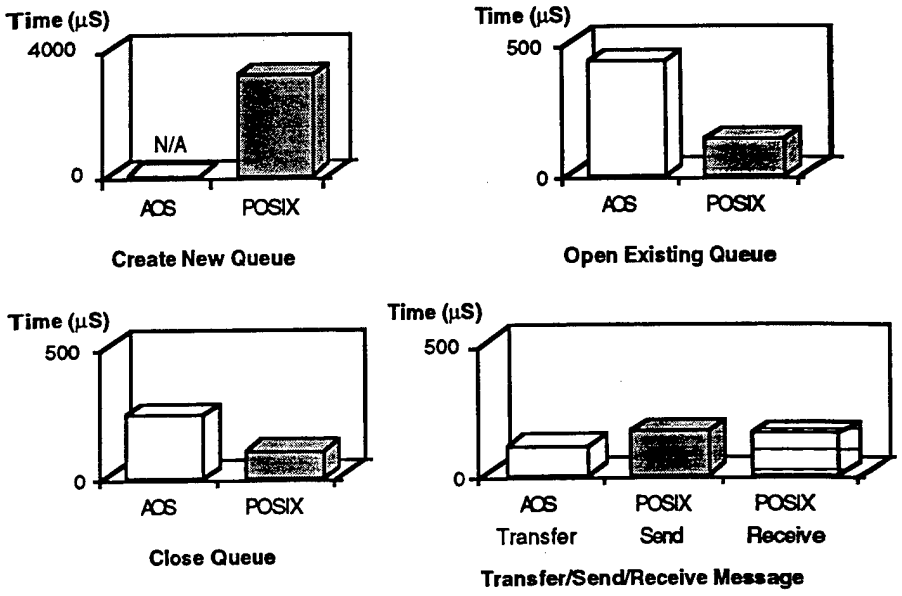



**Figure 2.4.1:** Message Queue Timings

### *Executable Size*

The size of the AOS Labeled Message implementation is 1.9 times the size of the POSIX Message Queue implementation. This is due to the additional functionality in AOS Labeled Messages. Labeled Messages provide several notification methods and transfer modes not present in POSIX Message Queues. The memory for the modules that provide interprocessor communications has been discounted from the AOS total, but some of this functionality is still present in the remainder of the code.

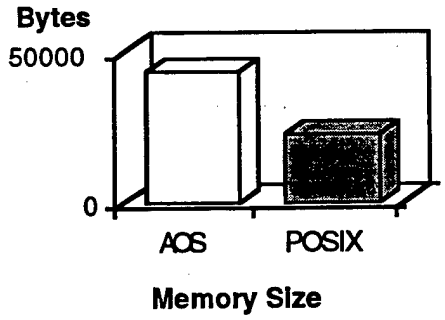



**Figure 2.4.2:** Message Queue Executable Size

### *Complexity*

The SLOC count for AOS Labeled Messages is about 1.5 times the SLOC count for POSIX Message Queues. Again, this is due to the additional functionality of AOS Labeled Messages.

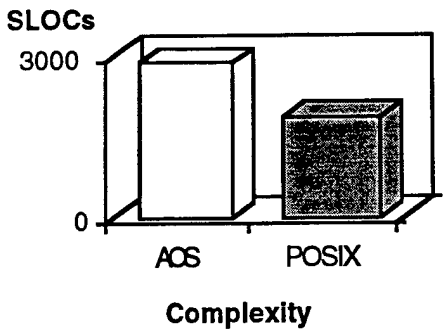



**Figure 2.4.3:** Message Queue Statement Lines of Code (SLOC)

### *Security*

Message passing interfaces have obvious security implications, since their only function is to pass data. Since POSIX Message queues must be opened or created before they can be accessed, its is a relatively simple process to add the necessary access privilege checks during the opening/creation of the message queue. However, message queue descriptors must follow the same restrictions that semaphore descriptors do; namely that a message queue descriptor is only valid within the process that originally received it from the OS. Failing to constrain the passing of message queue descriptors would allow for the possibility of opening an illicit high bandwidth data channel.

### *Implementation Difficulties*

A problem with POSIX Message Queues is that the current Ada binding restricts the possible implementations of message queue descriptors despite the descriptor being a private type. The `Set_Attributes` operation specifies that the descriptor is an "in" parameter only, which forces attributes to be stored somewhere other than in the descriptor.

# 3.0 Summary & Recommendation

## Summary

### *Technical Findings*

The general findings of this study are that some of the POSIX services have acceptable service times such that they can be used in a real-time avionics system. However, there are some issues with the long latencies introduced by the use of signals for asynchronous notifications.

In general, the POSIX services tend to occupy more memory than their AOS counterparts. This is due to additional functionality, much of which may not be useful in a real-time avionics system.

The POSIX services lack the security features needed in a multi-level secure environment. Adding the needed security features is a non-trivial undertaking. Specifying the security aspects of a specific system is implicitly difficult. Specifying a generic set of security features is even more difficult as previous attempts at adding security features to POSIX have shown. (So little progress was made by the POSIX security Working Group that in October of 1995, a motion was made to consider removing their Project Authorization Request).

A major piece missing from POSIX is a set of real-time distributed IPC (Inter-Process Communication) interfaces. There is a POSIX working group dedicated to producing a standard in this area, but they have only recently produced their first cut at a set of language bindings; it may be several years before a usable standard is approved.

The findings of this study have attempted to point out generic aspects of the POSIX Ada bindings that will hold true in most, if not all, implementations. However, when interpreting these findings, it must be remembered that this is only one implementation of the POSIX Ada bindings. Many aspects of the target system and of the tools used to implement the binding can affect the timing and sizing of an implementation. The suitability of using a POSIX implementation in a specific system must still be determined; simply using an implementation that is compliant to the standard is not a guarantee of suitability in of itself.

### *Other Areas of Concern*

This report has thus far focused only on the technical aspects of the use of the POSIX Ada bindings. However, there are other potential pitfalls of using POSIX and Ada.

- Vendor Support for POSIX

Very few of the shrinking number of Ada vendors are supporting bindings to POSIX. All commercial OS vendors are targeted to the C language market, and so they tend to support only a C binding if they support POSIX at all. Those who need an Ada binding to the POSIX may therefore be forced to write their own or pay someone to create one for them.

- Flagging Support for POSIX

Attendance at the POSIX working group meetings has been steadily shrinking for the last 2 years. Without support from interested parties (in the form of people to work on the standards), there will be no forward movement of the POSIX standards.

- Ada 95 And POSIX

It is an open question whether an Ada 95 compiler that fully supports the Systems and Real-Time Annexes obviates the need for a POSIX Ada API, since many of the functions of the POSIX API are met through the specification of the Ada 95 language itself. For example, semaphore services might be replaced by Ada 95's protected objects or packages Ada.Asynchronous_Task_Control and Ada.Synchronous_Task_Control.

## Recommendations

1) **Implement more services.**

During this study we have implemented a small subset of POSIX services: Semaphores, Timers, Signals, and Message Queues. We have identified issues and gathered data only made possible by implementing a POSIX interface in a real system. Therefore, we recommend that work be continued to implement more of the POSIX features specified in the real-time profiles. Some of these POSIX features are:

- Mutexes/Condition Variables: Synchronization mechanisms that include priority inheritance and priority ceiling mutexes. This would be helpful to implement if we pursue a Project Authorization Request (PAR) to include Condition Variables as an additional asynchronous notification mechanism, since this will allow us to have a working prototype.

- Process Creation/Termination: Services to create new processes (i.e., Ada programs or active partitions) and return status on their termination.

- Shared Memory/Memory Protection: An issue in a secured system. We need to implement allocation/management of shared memory, implement the setting of page protections, and identify issues.

- Task Scheduling: An implementation of the Ada 95 scheduling model. This task would require support from a compiler vendor since Ada pragmas are involved.

- Synchronous & Asynchronous I/O: I/O in step with and in parallel with program execution. Synchronous I/O calls are: open, close, read, write; basic I/O services. The asynchronous calls allow reads/writes to proceed in parallel with program execution. (Synchronous I/O is required first, since you must open a file/device before submitting read/write requests.) Implementing I/O requires a decision on the interconnect bus type (1553, PI-Bus, SCI, etc.).

- Synchronized I/O & Prioritized I/O: Features that may or may not be needed in an avionics environment. All AOS I/O is already synchronized (hence, the AOS does not do file buffering.). As for prioritized I/O, the AOS defines the I/O priority to be the priority of the process (i.e. the Ada program or active partition) and not the priority of the task/thread. All tasks in a program submit their requests at the same priority unless they actively request to submit at a lower priority.

- File Descriptor Management: Includes file locks, position seeking, and file descriptor duplication.

- Other: All of the more-or-less trivial packages to implement. If we reach this point, we will feel confident in the feasibility of using POSIX in a Real Time Secured Avionics Environment. This category includes system configuration packages, calendar services, environment services, pipes, memory locking (meaningless on systems without memory page swapping,) and other utility packages.

**2) Define POSIX security.**

During this implementation of the POSIX features on the AOS, security shortcomings have been identified. There is a need to develop and recommend modifications to those POSIX functions to support security. Our implementation demonstrates that a Security Profile is needed and a task should develop this Security Profile.

**3) Participate in the POSIX Working Groups.**

We have found issues with POSIX that we have brought up to the POSIX Working Group. In particular, we have identified the need for the standard to allow different types of event notification besides POSIX signals. At the January, 1996 meeting of the POSIX Working Group, we intend to request that Semaphores or other simpler mechanisms be allowed for event notification

**4) Study the Impact of Ada 95 on POSIX.**

What features in Ada 95 can replace OS services? What support must an OS have for Ada 95? Are they necessary and what form should the POSIX bindings for Ada 95 take?

## 4.0 References

*Standard for Information Technology —Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API), Amendment 1: Real-time Extension [C Language]* . IEEE Std. 1003.1b-1993, herein referred to as POSIX.1b.

*Standards Project Draft Standard for Information Technology—Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API), Amendment #: Protection, Audit and Control Interfaces [C Language].* IEEE P1003.1e Draft 14 March 1994, herein referred to as POSIX.1e.

*Standards Project Draft Standard for Information Technology —POSIX Ada Language Interfaces—Part 2: Bindings for real-time Extensions.* IEEE P1003.5b, Draft 5.0 May 1995, herein referred to as POSIX.5b.

*Standards Project Draft Standard for Information Technology—Standardized Application Environment Profile—POSIX Real-time Application Support (AEP).* IEEE P1003.13 Draft 7.0 August 1995, herein referred to as POSIX.13.

*POSIX Delta Document For the Next-Generation Computer Resources (NGCR) Operating Systems Interface Standard Baseline (Version 4)*, Operating Systems Standards Working Group (OSSWG), Report N° NAWCADWAR-94109-70, 1 June 1994